PSPSOCK
Prospero Software Products - Sockets System

User Guide
Version 3.2
July 2011

ABOUT THIS DOCUMENT

PSPSOCK is a software framework you use to develop application software solutions based on TCP/IP sockets for the IBM ISeries platform.

This document does not cover low-level TCP/IP sockets technology. Other guides available from our Web site - and other Web sites - do that.

PSPSOCK is licensed technology. You install the PSPSOCK library on your ISeries machine(s). You use the objects in the library to develop:

- Sockets-based services, where each service comprises several ISeries server jobs. These services run only on ISeries.
- Sockets-based client systems. The distinction between ‘client’ and ‘server’ is fundamental to sockets applications. The client systems may run on your ISeries or - more typically - another platform.

PSPSOCK provides facilities to develop new services and operate existing services e.g. start, stop and monitor the services.

This user guide is written primarily for developers who want to use PSPSOCK to develop new services. It provides example code mainly in ILE/RPG. Many of the core modules in the PSPSOCK library are written in ILE/C. You do not have to know ILE/C to use PSPSOCK. PSPSOCK shields you from the complexity of low-level sockets communications.

PSPSOCK LICENSING

The PSPSOCK library contains two data areas:

- PSPSLICKEY. Contains the required 30-character license key.
- PSPSUSR: Contains your company name. Maximum 30 characters. This name is not essential for PSPSOCK processing, but we recommend you update the data area anyway.

The license key is fairly cryptic e.g. is F7QOQ83SZCGABDLQ94K537WZ6G4X7O on our own ISeries machine. It varies per CPU. That is, each ISeries machine with its own serial number must have a specific license. You can see your serial number using the DSPSYSVAL SYSVAL(QSRLNBR) command.

As part of a free trial you can get a temporary license number. Instructions on how to do this are on our Web site.

When you are approaching the end of your license period, you start getting CPF9898 messages to the *SYSOPR message queue. These messages look like this:

```
WARNING: The license for the PSPSOCK library on this machine (serial
  number XXXXXX) expires in 2 days. All sockets functions in the PSPSOCK
  library will disable shortly. See http://www.prosperosoftware.co.uk.
WARNING: The license for the PSPSOCK library on this machine (serial
  number XXXXXX) expires tomorrow. All sockets functions in the PSPSOCK
  library will disable shortly. See http://www.prosperosoftware.co.uk.
ERROR: The license for the PSPSOCK library on this machine (serial number
  XXXXXX) has expired. All sockets services in the PSPSOCK library are
  disabled. See http://www.prosperosoftware.co.uk.
```

These messages start getting sent 2 days before your license actually expires. They are only sent when you try to start up a PSPSOCK service. To prevent a flood of messages to *SYSOPR, the messages are only sent once per service per day. In addition a service log record is created - again only once per service per day. You can see these records for each service by running the PSPSOCK command then taking option 13=Messages for each service.

The intent behind this licensing mechanism is simply to enforce our licensing terms. If you get the licensing wrong, we react quickly to ensure that PSPSOCK services continue to run to support your business. If you have issues, for example, when you migrate from one ISeries machine to another or start using partitions then we are happy to help. In cases like this we suggest you contact us as early as possible.

The PSPSOCK library and upgrades

PSPSOCK library contents

The PSPSOCK library as delivered to you is quite small - about 10MB. It contains the following objects:

1) Two data areas PSPSLICKEY and PSPSUSR - discussed in previous section of this document.
2) Programs that support the PSPSOCK and PSPOPRSCK commands. You use these commands to run and administer PSPSOCK. See the next section of this document.
3) Two database files PSPSCTL and PSPSLOG that hold details of PSPSOCK services and their logged messages. Fields in these files are discussed in detail later in this document.
4) The PUSRMSG message file. This contains all messages used by the system. You see the message text on PSPSOCK control screens.
5) Program modules, binding directories and RPG/LE source files. You use these to build programs that support new sockets-based services for your applications.
6) Locking data areas for the two demonstration sockets services we provide.

PSPSOCK library upgrades

If you upgrade your version of PSPSOCK, you receive a new version of the whole PSPSOCK library from us. You then need to manage the migration away from your old version of the PSPSOCK library to the new version. We recommend you rename the old version of the library (e.g. from PSPSOCK to PSPSOCK2), restore the new version of your library to your system them migrate as described below. During the migration you should ensure PSPSOCK is not in use.

1) Copy the PSPSLICKEY and PSPSUSR data areas from PSPSOCK2 to PSPSOCK.
2) Copy the PSPSCTL file from PSPSOCK2 to PSPSOCK using CPYF MBROPT(*ADD) FMTOPT(*MAP *DROP) ERRLVL(*NOMAX). This preserves the few records in the file for the default services we provide and appends records for any further services you previously defined.
3) Copy the PSPSLOG file from PSPSOCK2 to PSPSOCK using CPYF MBROPT(*REPLACE) FMTOPT(*MAP *DROP). This preserves any log messages for services - for both our demonstration services and your own services - as they have run on your system.
4) Compare the members in PSPSOCK/QRPGLESRC and PSPSOCK2/QRPGLESRC. If you have added your own members in the latter file (something we do not recommend), you need to reinstate them in the former file using appropriate CPYF operations. However you do this, it is vital that you do not override the members we provided in the new PSPSOCK library.
5) Compare the PSPSOCK and PSPSOCK2 libraries to see if there are any other objects in the PSPSOCK2 that are not in PSPSOCK. You might see, for example, locking data areas or modules and programs for services you have developed yourself. Copy/move those objects from PSPSOCK2 to PSPSOCK.

It considerably simplifies this migration if you do *not* add your own objects or source members into PSPSOCK. If you respect this, then the only items you need to migrate are 1) to 3) above.

PSPSOCK OPERATIONS

The PSPSOCK command

The PSPSOCK command brings up a display that shows current services. You can start, stop and view logged messages for each service. You may have up to 99,999 services. PSPSOCK as initially installed on your ISeries has 2 services pre-defined. You add more services as described in following sections of this document.

The PSPSOCK command has the following syntax: PSPSOCK ALWCRT(*YES) ALWCHG(*YES) ALWSTR(*YES) ALWEND(*YES) ALWHLO(*YES). You can change any of the keyword values from *YES to *NO. The keywords have the following functions:

- ALWCRT(*YES): Enables the option F6=Create on the PSPSOCK display.
- ALWCHG(*YES): Enables the option 2=Edit on the PSPSOCK display.
- ALWSTR(*YES): Enables the option 1=Start on the PSPSOCK display.
- ALWEND(*YES): Enables the option 9=End on the PSPSOCK display.
- ALWHLO(*YES): Enables the option 15=Hello(Ping) on the PSPSOCK display.

The PSPOPRSCK command

The PSPOPRSCK command also provides the ability to manage services if you want to do this in your own CL program rather than using the user interface the PSPSOCK command provides. The command generates messages as defined in the PUSRMSG message file. The PSPOPRSCK command has the following syntax: PSPOPRSCK SERVICE(1-99999) OPER(*START/*END/*HELLO) MSGOPT(*LOGF/*JOB/*BOTH/*NONE).

- OPER(*START): Attempts to start the service. Generates message PSP0033 if the service started successfully; PSP0039 if the service controlling data area was not found; PSP0040 if the controlling data area already has object locks.
- OPER(*END): Attempts to end the service. Generates message PSP0036 if the service ended successfully; PSP0035 or PSP0037 if the service was not able to be ended.
- OPER(*HELLO): Tests if the service responds to sockets communications - using an internal proprietary application protocol. Generates message PSP0041 if the service responded successfully; PSP0038 or PSP0042 if the service did not respond.
- MSGOPT(*LOGF): Messages in response to running the PSPOPRSCK command are written to the PSPSLOG file. You can view these by running the PSPSOCK command then option 13=Messages.
- MSGOPT(*JOB): Messages in response to running the PSPOPRSCK command are sent as escape messages to your program message queue e.g. so you can monitor them in a CL program using the MONMSG command.
- MSGOPT(*BOTH): Messages are sent to both your program message queue and the PSPSLOG file.
- MSGOPT(*NONE): No messages are sent.

Note that PSPOPRSCK OPER(*START) submits a sockets control job to a job queue. The PSP0033 message is only generated by the batch job when it starts running. So the second MONMSG in the sample CL program code below is meaningless:

```
PGM
PSPOPRSCK SERVICE(&SERVICE) OPER(*START) MSGOPT(*JOB)
MONMSG MSG(PSP0039 PSP0040) EXEC(DO)
/* Valid test for service start failed */
ENDDO
MONMSG MSG(PSP0033) EXEC(DO)
/* This test for service started successfully will never be invoked */
ENDDO
ENDPGM
```

Instead you can start the service then test it using the '*HELLO' operation as in the following sample code:

```
PGM
PSPOPRSCK SERVICE(&SERVICE) OPER(*START) MSGOPT(*JOB)
MONMSG MSG(PSP0039 PSP0040) EXEC(DO)
/* Valid test for service start failed */
ENDDO
/* Wait a few seconds for the submitted job to start running */
DLYJOB DLY(5)
PSPOPRSCK SERVICE(&SERVICE) OPER(*HELLO) MSGOPT(*JOB)
MONMSG MSG(PSP0041) EXEC(DO)
/* Valid test for service running */
ENDDO
MONMSG MSG(PSP0038 PSP0042) EXEC(DO)
/* Valid test for service not running */
ENDDO
ENDPGM
```

Developing new services

Application protocols

The client and server for any new service need to agree an application protocol. This defines the data they exchange. We provide 2 demonstration services with PSPSOCK.

PSPSOCK service number 1 is an 'echo' service. The server returns verbatim the data it receives from the client system. The application protocol here is very simple.

PSPSOCK service number 2 is a 'currency description lookup' service. It has the following protocol:

- The client system sends an ISO currency code ("USD", "JPY" etc.) to the server, along with a single character indicating if the supplied value is EBCDIC or ASCII.
- The server system responds with the currency code plus description: "USDUnited States Dollar", "JPYJapanese Yen" etc.

You need to define your own application protocol for every service you develop. These protocols can be moderately complicated when you consider issues such as reliability, error handling, service availability and performance etc. The final section of this document provides some guidelines on protocol development.

Service development

Each new service you develop requires a new ISeries program. This program is what you define on the PSPSOCK screen as the *Worker program name*. To build this program, you need to develop at least one RPG/LE module, compile it using the CRTRPGMOD command, then create the worker program using the CRTPGM command.

Start by looking at the members in the source file PSPSOCK/QRPGLESRC. Please do **not** edit the default members provided. These have descriptions 'Base', 'Service 1' and 'Service 2'.

Assume you want to develop your own service number 10. The procedure to develop this new service is as follows:

1) Identify a unique port number for your new service. See the section Port number later on in this document.
2) Use the PSPSOCK command to define your new service. Do not use a service number less than 10. Those numbers are reserved for Prospero Software. For the new service, ensure that the following are unique: Lock data area and Control job name. You can follow the naming convention PSPSnnnnnS (where nnnnn is service number) for some names. For example, your first service number 10 could have PSPS00010S for the Lock data area and Control job name. Look at the definitions of the services 1 and 2 for guidance.
3) Create the lock data area. This can be just a single character in length. Look at the data areas defined for service 1 and 2 and consider copying one of them. See the section Lock data area further on in this document for more information. We recommend you create the data area in a library other than PSPSOCK.
4) Copy source member PSPS00000S in PSPSOCK/QRPGLESRC to new member e.g. PSPS00010S. Because this is your source code not ours we recommend you create the member in your own application library. Assume QGPL/QRPGLESRC in this example.
5) Modify the PSPS00010S source member code to support your own application protocol.
6) Run the following command to compile your new module: CRTRPGMOD MODULE(QTEMP/PSPS750) SRCFILE(QGPL/QRPGLESRC) SRCMBR(PSPS00010S). Note that the module name must be PSPS750. The module does not have to be created in QTEMP if you would prefer to use another library. We do not recommend you create the module in the PSPSOCK library.
7) Run the following command to create the new service program: CRTPGM PGM(QGPL/PSPS00010S) MODULE(QTEMP/PSPS750) ENTMOD(PSPS600) BNDDIR(PSPSOCK/PCRTSRVPGM) ACTGRP(*CALLER). We do not recommend you create the program in the PSPSOCK library. The *CALLER activation group is recommended but not compulsory.
8) Test the service using the PSPSOCK command or by running the PSPOPRSCK command. Use the PSPSOCK command to look at messages from the server for the service.

Client development

If the client programs that use your new service are running on non-ISeries platforms, they are likely to be written in languages such as C, C#, VB or Java. You may also choose to develop an ISeries ILE/RPG client program to test your new service. The procedure to develop this for your service number 10 is as follows:

1) Copy source member PSPS00000C in PSPSOCK/QRPGLESRC to new member e.g. PSPS00010C. Because this is your source code not ours we recommend you create the member in your own application library. Assume QGPL/QRPGLESRC in this example.
2) Modify the PSPS00010C source member code to support your own application protocol.
3) Run the following command to compile your new module: CRTRPGMOD MODULE(QTEMP/PSPS00010C) SRCFILE(QGPL/QRPGLESRC). The module does not have to be created in QTEMP if you would prefer to use another library. We do not recommend you create the module in the PSPSOCK library.
4) Run the following command to create the new client program: CRTPGM PGM(QGPL/PSPS00010C) MODULE(QTEMP/PSPS00010C) BNDDIR(PSPSOCK/PCRTCLTPGM) ACTGRP(*CALLER). The program does not have to be created in the PSPSOCK library if you would prefer another library. It will never be automatically invoked - you always have to run it manually. So there is no particular value in creating it in the PSPSOCK library. The *CALLER activation group is recommended but not compulsory.
5) Run the new program by calling it. Use the PSPSOCK command to look at messages - from both client and server - for the service.

PSPSOCK/QRPGLESRC contains RPG/LE source code for clients of PSPSOCK service number 1 (the ‘echo service’) and service number 2 (the ‘currency description lookup service’). These are in members PSPS00001C and PSPS00002C respectively. In addition, Java versions of these programs are available on our Web site.

Service control records

The PSPSOCK command shows the fields below for each sockets service. These are in the PSPSCTL file. This file is delivered in the PSPSOCK library. There is nothing in principle to stop you copying the PSPSCTL physical file - and the based on logical files - to another library, although we will obviously not be too interested in supporting you if you then fail to use the copied file correctly.

Authority to the PSPSCTL file in the PSPSOCK library is not restricted by default. You may consider this a potential risk and want to set up some file access/update controls of your own. We strongly prefer you to use the PSPSOCK command to edit records in the file rather than editing those records using a generic utility. The PSPSOCK command has options that restrict the ability to update PSPSCTL records.

Service number (Key)

The unique identifier of the service. Each service has a dedicated port number on which it listens and has dedicated programs that support the application service protocol. You may have up to 99,999 services in total. Most PSPSOCK users are unlikely to get beyond 100 services. Service numbers less than 10 are reserved for Prospero Software. When you add a new record, the key auto-increments by 1.

Service Version

The PSPSCTL file has built-in versioning. Every time you edit a service control record, a new version is created with the new details. The old versions are visible for each control record using option 15=Versions. You can thus see the editing history of the changes you have made for the service. Up to 99,999 versions may exist for each service.. The highest-numbered (most recent) version is the one used by PSPSOCK internal functions.

Name

Your own 40-character description of the service. May be blank.

Port number

The IP port number this service listens on. 1 to 99999. Must be unique for each service. In practical terms you should avoid the so-called “well known port numbers”. If you search for *well known IP ports* in any decent Internet search engine, you should find sites like IANA (www.iana.org) that cover good practise for assigning your own port numbers. Note that client processes need to able to connect to those ports. If the client process is running on a platform other than your ISeries, you need to check that network devices like intermediate firewalls are not blocking access to the ports you use. Any change to this value comes into effect when you next start the service.

Control job description

The name of the AS/400 job description under which the service process will run. Defaults to *USRPRF if you leave it blank. See values on the SBMJOB command for the JOBD keyword. Any change to this value comes into effect when you next start the service.

For the control job description library, you may specify a library name or values *LIBL or *CURLIB.

Control job queue

The name of the AS/400 job queue to which the service process will be submitted to run. Defaults to *JOBD if you leave it blank. See values on the SBMJOB command for the JOBQ keyword. Any change to this value comes into effect when you next start the service.

For the control job queue library, you may specify a library name or values *LIBL or *CURLIB.

Control job name

The name of the AS/400 service process job. We recommend you adopt the naming convention PSPSnnnnnS where nnnnn is the service number e.g. PSPS00026S would be the name of the control job for the 26th service you define. This is the default if you blank out the value. Apart from your own explicit value, you may also use the value *JOBD. See values on the SBMJOB command for the JOB keyword. Any change to this value comes into effect when you next start the service.

Control job user

The name of the user profile under which the AS/400 service job runs. Defaults to *CURRENT if blanked out. Apart from a specific named user, the *JOBD value is allowed. See values on the SBMJOB command for the USER keyword. If you have a named user profile, the user trying to start the service must be authorised to the named user profile. If you encounter a problem with this, there are various solutions - mostly with programs adopting authority. Contact us for advice if you are not certain what to do. Any change to this value comes into effect when you next start the service.

Worker job name

When the service starts running, one control process (with *Control job name*) starts running. It immediately starts up 1 or more worker jobs that provide the core of the service. These worker jobs inherit most of the run-time attributes of the control job. To distinguish the worker jobs from the control job, you can give them different names. We recommend you adopt the naming convention PSPSnnnnnW for the worker jobs where nnnnn is the service number e.g. PSPS00026W would be the name for the 26th service you define. This is the default if you blank out the value. Any change to this value comes into effect when you next start the service.

Important: This facility is not yet implemented in PSPSOCK. The *worker job name* will for the time being always be set to the *Control job name*. This is an API issue we hope to sort out in a future release of PSPSOCK.

Worker job user

The name of the user profile under which the AS/400 worker job runs. Can be left blank - in which case the worker jobs run under the same user profile as the control job - see *Control job user* above. Any change to this value comes into effect when you next start the service.

Important: This facility is not yet implemented in PSPSOCK. The *worker job user* will for the time being always be set to the *Control job user*. This is an API issue we hope to sort out in a future release of PSPSOCK.

Lock data area

When the service starts running, the control job places an *EXCLRD lock on this data area, while each of the worker jobs places an *SHRRD lock on this data area. By examining these locks, PSPSOCK is able to determine which processes are running. You should avoid deliberately placing your own long-term locks on this data area. You obviously can not stop this completely e.g. a backup you run will place a very short-term lock on the data area.

The type and contents of the data area are immaterial. It can be defined as TYPE(*CHAR) LEN(1). We recommend it has the same name as *Control job name* above i.e. PSPSnnnnnS where nnnnn is the service number. We do not recommend you create these data areas for your own services in the PSPSOCK library - please use one of your own libraries.

You can not change this value using the PSPSOCK command if there are currently any locks on the data area e.g. the service may currently be running. Any change to this value comes into effect when you next start the service.

For the lock data area library, you may specify a library name or values *LIBL or *CURLIB.

Log file library

For the log file library, you may specify a library name or values *LIBL or *CURLIB. Every PSPSOCK service will log messages to a log file PSPSLOG. This file is provided as standard in the PSPSOCK library. You can copy this file (and based-on logical files) to another library if you wish. In which case you can update the log message file library using PSPSOCK. Any change to this value comes into effect when you next start the service.

Note that this value only applies to the service running as a batch job, not to the PSPSOCK and PSPOPRSCK commands. Those commands will use the PSPLOG file in their library list. If you want to log PSPSOCK messages consistently to a PSPSLOG file other than the default one provided in the PSPSOCK library, you need to manage your library lists carefully. You may even consider deleting, renaming or moving PSPSOCK/PSPSLOG and the logical files based on it.

Worker program name

You must develop your own unique worker program for each service. This worker program implements the application protocol for that service. See previous section *Client development*.

Audit level

PSPSOCK offers various levels of logging - to help you monitor what each service is doing and any potential problems with that service. Any change to this value comes into effect when you next start the service. This variable has values as follows:

- 0 (zero): The only logging is error messages to the PSPSLOG file. You can see those using option 13=Messages from the PSPSOCK command.
- 20: Same as 0 (zero), but additional messages are logged to PSPSLOG every time the service starts and ends normally. This is the default for new services.
- 50: Same as 20, but the control and worker service jobs generate a spool file documenting key phases in their life cycle e.g. when each worker job interacts with a client process.
- 70: Same as 50, but the control and worker service jobs provide a very detailed audit of bytes transmitted to/from client processes. We recommend you use this option only when advised by us. The logging process with this audit level introduces significant overhead.

Number of worker jobs

Each service allows between 1 and 999 worker jobs to run. Any change to this value comes into effect when you next start the service. The worker programs provide the support for the application protocol specific to that service. See *Worker program* above. There is no iron-clad rule as to the initial value for this setting. Generally speaking, you should start out by setting this value to a reasonable number e.g. 5-20 then increasing it gradually only when you have to. To see if any one job is busy, use DSPJOB to monitor its CPU usage over time or run WRKTCPSTS OPTION(*CNN) and for the port the service listens on take option 8=Display Jobs. In this way you can see if you have too many worker programs running. Too few worker programs you can simply detect by the entire service being slow or timing out altogether on client systems.

The performance of any one job is highly dependent upon your application protocol. You can get a feel for the raw performance of the PSPSOCK framework using service number 1 - the ‘echo service’.

Audit Date/Time

Every service record is automatically stamped with the date/time it was first created in CCYYMMDD HHMMSS format. This is not the date/time it was updated - because no updates ever occur after the record was written. A new version record is always written instead.

Service log records

The PSPSOCK command shows logged messages for each sockets service. You can see those using option 13=Messages from the PSPSOCK command. These are messages in the PSPSLOG file. This file is delivered in the PSPSOCK library. There is nothing in principle to stop you copying the PSPSLOG physical file - and the based on logical files - to another library, although we will obviously not be too interested in supporting you if you then fail to use the copied file correctly.

Authority to the PSPSLOG file in the PSPSOCK library is not restricted by default. You may consider this a potential risk and want to set up some file access/update controls of your own. We strongly prefer you to use the PSPSOCK command to view records in the file. We do not recommend you edit individual records in this file using a generic utility. If the file over time gets a very large number of records, you may want to drop historical records that are no longer of any interest. We have no problem with that provided you do it sensibly e.g. write your own program that deletes individual records created before a certain date/time. We do not recommend bulk CPYF operations to remove unwanted records, particularly if any services are currently running.

Service number (Key)

The identifier of the service. See the same field in the preceding *Service control records* section.

Service Version

The version of the service. See the same field in the preceding *Service control records* section.

Timestamp

The date/time - to the nearest millisecond - that the log record was created in ISO format i.e. the date portion is CCYY-MM-DD. The logical file PSPSLOG1 shows records for each service in descending (most recent first) sequence. This is the view you get when you take option 13=Messages for a service using the PSPSOCK command.

Program name

The name of the *PGM object that generated the log message. These programs are ILE programs with many modules.

Job number/name/user

Standard ISeries job identifiers. You can see this by running the DSPJOB command for the PSPSOCK server job.

Message ID

The 7-character message identifier. All messages can be seen in the PUSRMSG file in the PSPSOCK library.

Message Severity

The severity of the message in range 00-99.

Message data

This field may contain variables fields that appear in the message (in addition to *Service number* and *Service version*). If we assist you with a PSPSOCK problem, we may ask you to provide us with the contents of this field - it can provide valuable clues to the reason for an underlying problem that generated the message.

Message text

The expanded 1st-level message text from the PUSRMSG file with variables substituted.

Sockets application protocols

It is not the purpose of this User Guide to deal with all issues regarding sockets-based application protocols. However this section offers some guidelines. We also recommend you look at the "Sockets overview for IT Managers" document (psp_iseries_sockets.pdf) available on our Web site.

Data length

PSPSOCK requires both client and server to specify the length of data they are sending to each other. This is in addition to the actual data which is subject to your protocol. These lengths are available to your application-specific program code. See example RPG/LE source code for the WRDTA_LEN and RDDTA_LEN variables. PSPSOCK will always transmit exactly these lengths. If for some reason it is unable to do so, the service will fail with a low-level error message to the PSPSLOG file.

You could program client and server to work with a fixed minimum length e.g. send/receive data from 2K internal buffers and always set 2048 for data length. However the length of data may be variable within one protocol e.g. to support multiple transactions or different versions of the same transaction. It may also be truly variable for one transaction. For example, the service is returning details for between 1 and 10 clients - in which case the count (1-10) should probably also be somewhere in the protocol itself.

On balance there is no harm in setting as exact values for WRDTA_LEN and RDDTA_LEN as your protocol allows. It is useful information for your application program code.

Transaction identification

Even if your service supports only one transaction, it is a good idea to allow for a transaction ID in your protocol. This could be as simple as 2-byte numeric value where the value 1 represents your 1st transaction, 2 represents your 2nd transaction or the 2nd variant of your first transaction etc.

Message identification

Related to transaction ID, but at a lower level. For example if the server processes transaction 1 successfully, it returns 0 as the message ID. If the server has any problem, it returns a different message ID: 1 if the service is unavailable at the moment, 2 if the server can not understand the request etc. In short, the client needs to distinguish error conditions from normal server responses.

It is possible to use ISeries messaging standards here e.g. SSV0000 for normal service, SSV0001 for service error 1 etc. This provides good documentation of message text and variable substitution.

Session identification

The simplest case is where a client system connects to the server and gets all the data it needs in one hit without any need for client authentication. If you introduce client authentication (the client must prove that it is trusted) and/or have multiple-hit design (the client initiates a transaction in one hit then returns later and gets the transaction data), then you are into "session" issues. This typically involves the exchange of some kind of session identifier between client and server. The client signals a new session and the server returns the session ID. The client then returns this session ID on subsequent hits until it either explicitly signals end of session or simply discontinues use of that session ID - in which case the server should probably expire the session ID after a suitable interval of time.

Character set issues

The data exchange via sockets is simply streams of bytes. If you need to convert between different character sets (e.g. between an ASCII-based client and an EBCDIC-based server) you have to do it yourself. It makes sense that the server does this, not the client. The demonstration service 2 (the 'currency description lookup service') does this using the QDCXLATE API.

You need to design this translation to work only on character fields. For example if the data returned to the client is built in an RPG/LE data structure, you can populate the structure with character fields, translate the entire data structure, then populate the remaining numeric fields.

File open/close and cacheing

Frequently-used files can remain open on the server between client hits - to save the significant overhead of file opens. You may want to qualify this e.g. if you have some concern about long-running object locks on files you are trying to include in a backup. You can issue instructions via the socket to the server e.g. instruct it to switch between "keep files open" and "close files after use" modes.

You can control cacheing in a similar way. Some server data may change infrequently - currency codes and descriptions are a good example. A server using those could be designed to read all currency codes/descriptions into internal arrays on startup then only reload those arrays when instructed to do so via a socket connection.

Logging

You are likely to want some internal logging in your servers - if only to track the number of client hits over time. This needs to be reasonable i.e. the overhead of logging should not significantly impact the rest of the service.

Operational instructions

Server instructions such as ‘open/close files’, ‘reload arrays’, ‘advise service temporarily unavailable’, ‘increase/decrease logging level’ are best advised to the server via the socket connection rather than in another way - lookup a data queue / data area / control database record etc. You just need to include these instructions in your sockets protocol. The instructions are typically issued by another program on the server. You want to make it very unlikely that a client program could accidentally issue these instructions. In the simplest case, only accept those instructions if they include a fairly long ‘magic number’ that is not easily obtainable.

This bit of application code is reusable. Once you have built it for one service, it can be plugged in to subsequent services.

Security and encryption

If you have a concern about rogue clients accessing the server, your first line of defence is control of open ports by firewalls and other network devices. Typically these have ‘allow selected ports, deny all other ports’ configurations. Beyond that, it is certainly possible to build some kind of security into your application protocol e.g. require passwords and have fairly complicated session management. This work is non-trivial. If you want to do it properly, you need to examine every potential point of weakness. For example if you introduce passwords, you must consider where and how (encrypted or unencrypted) those passwords are stored, how frequently the passwords expire, how the server detects expiry etc.

In practical terms, there is likely to be a degree of reliance on ‘trusted clients’ particularly if all socket connections are known to come from a highly-trusted area of your network. If that network security is breached, all bets are off anyway - the sockets services are compromised to the same extent as any other systems/databases you run.